14/03/24 — Synchronization

(1) Data synchronization due to data inconsistency.

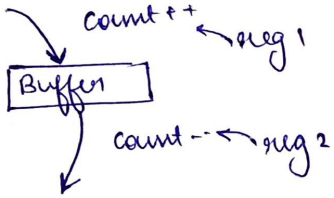
race condition ⇒ to one to execute last
↳ to & to avoid data inconsistency.

1) Mutual exclusion ⇒ mutually exclusive access.
at a time only one process can work.

2) progress ⇒ some process should work.

3) Bounded wait ⇒
wait period is limited.

↳ Three conditions must be there while performing synchro)



reg ← count
reg ← reg + 1
count ← reg

⇒ critical section ⇒ when process is working on the shared resource.
↳ ensure only one process should work.

⇒ entry section ⇒ allowing processes can work on critical section

⇒ exit section ⇒ process exits.

⇒ remainder section ⇒ remaining section of the process.
processes are not allowed to make to critical section.

⇒ Maintaining Synchronization :-

⇒ Software based ⇒ Peterson's Algo
At a time only one process should access the critical section.

do {

entry section [flag[i] = = true (willing → Pi)
turn = j . → chance to Pj
while (flag[j] && turn = = j);

C.S

exit section [flag[i] = = false] CS

R.S

} while (true)

// flag = true if process is willing to entry critical section

// turn value can be i and j if there are two processes Pi & Pj.

Hardware Solutions :-

bool test_and_set (boolean * target) (✓ lock)
{
boolean rv = * target ;
* target = true ; // forcefully true, wait
return rv ;
}

Initially lock = 0 / false
↓
wait khtm

do {
while (test_and_set (& lock)) ;] entry section.
CS
lock = false .
R.S. }
while (true)

→ in order to ensure 100% synchronization.

compare_and_swap (int * value, int expected, int new value)
(expected: false; new value: true)

{ int temp = * value ; (initially)
if (* value = = expected)
* value {= new_value) ;
}
return temp ;
}

while ((compare and swap (& lock, 0, 1)) ;

dusro ke liye lock disable kar rhe hai and after execution enabling the lock.

Semaphore → It is used to maintain sync. for multiple processes.
↳ to access it → wait (s) // exactly like while to move processes to wait
↳ signal (s)

```
wait (s) {
    while ( s <= 0) ; // do nothing
        s-- ;   // to keep the process suspended.
}
signal (s) {
        s++ ;
}
```

```
P1                                          P2
wait (s); // no other process can          wait (Q);
          use semaphore s.                 wait (s);
wait (Q);
   :       ——————→ deadlock ←——————            :
signal (s);                                signal (Q)
signal (Q);                                signal (s)
```

Representative Problems that represent different scenarios

① Bounded Buffer :-

semaphore mutex = 1 ← ensures mutual exclusion.
" empty = n
" full = 0

```
do {
    wait (empty);
    wait (mutex);
    signal (mutex);
    signal (full);
} while (true);
```

```
do {
    wait (full);
    wait (mutex);
    signal (mutex);
    signal (empty);
} while (true);
```

(2) The Readers Writers Problem :-

↳ Multiple readers can be there but no multiple writers and no simultaneous reader and writer.

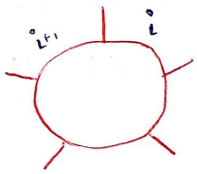
```
Semaphore rw_mutex = 1 ;        → mutual exclusion.
Semaphore mutex = 1 ;           → to exclusively update the read count
int read_count = 0;
```

```
do {
    wait (rw_mutex);
    signal (rw_mutex);
} while (true)
```

Writer process

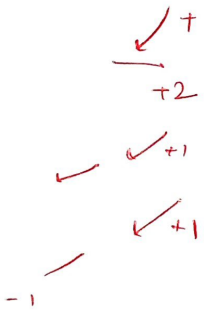
```
do {
    wait (mutex)
    read_count ++ ;
    if (read_count == 1)
        wait (rw_mutex)
    signal (mutex);
    wait (mutex);
    read_count --;
    if (read_count == 0)
        signal (rw_mutex);
    signal (mutex);
} while (true)
```

Reader process



The Dining Philosopher Problem :-

We need one semaphore exclusively for each chopstick




```
Semaphore chopstick [5] = 1
do {
    wait (chopstick [i]);
    wait (chopstick [(i+1) % 5]);
    signal (chopstick [i]);
    signal (chopstick [(i+1) % 5]);
} while (true);
```

deadlock

↳ hold and wait.

⇒ **Monitors**

↳ to ensure wait and signal are placed at proper position

↳ collection of functions in which we have diff semaphores.

Monitor DP {

```
enum { Thinking, Hungry, Eating } state[5];
condition self[5];
void pickup (int i) {
    state[i] = HUNGRY;
    test(i);        // to check if both chopstick are available.
    if (state[i] != EATING)
        self[i].wait();
}
void putdown(int i) {
    state[i] = Thinking;
    test((i+4)%5);
    test((i+1)%5);
}
```

}

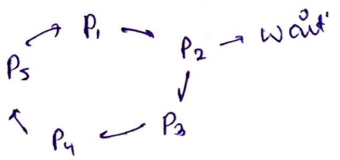
Deadlocks

No particular process is under execution when the set of resources are being held by other processes.

Processes can work over the resources after an request from the OS

⇒ Necessary conditions for deadlock to occur.

- ⇒ mutual exclusion ⇒ resources are non shareable at a time only one process mutually exclusive access to the resources
- ⇒ hold and wait → has some of the resources and is waiting for other resources.
- ⇒ No preemption
- ⇒ circular wait. → set of resources are held by processes and no release of processes.



Deadlock check.

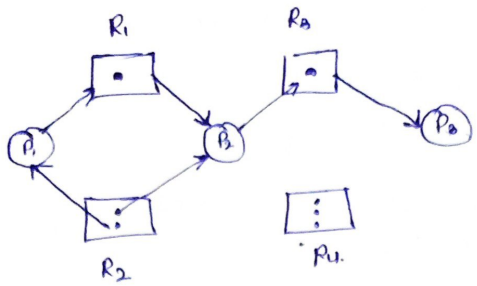
⇒ We can use resource allocation graph.

Set of resources $R = \{R_1, R_2 \dots R_m\}$.

Processes $P = \{P_1, P_2 \dots P_n\}$

edges - Request edge $P_i \rightarrow R_j$ // P_i is requested for R_j

Assignment edge $R_i \rightarrow P_j$ // R_j resources are held by P_j.



// Dots are instance of that particular process.

// Circular dection may ensure deadlock is there.

Deadlock Handling

→ Deadlock Prevention → in the beginning of the process
→ Deadlock Avoidance → during execution, runtime
→ • Detection and Recovery →

- Deadlock Prevention provides the set of methods to ensure that atleast one of the necessary condition cannot hold.

[These methods prevent deadlock by constraining how the request for the resources can be made *]

- Deadlock Avoidance requires that OS be given additional info in advance regarding which resources a process will request and use during its lifetime. With this knowledge OS can decide for each request whether or not process should wait. *

- To remove hold and wait in Deadlock Prevention we can use whenever a process request a resource i.e does not hold any other resource.

⇒ One protocol we can use each process to request and be allocated all its resources before it begins its execution.

⇒ Alternatively ⇒ We can allow a process to request only when it holds the thing. A process may request some p resources use them before it can request additional resources it must release all resources it has been allocated.

⇒ If a process is holding some resources and request another that cannot be allocated to it then all the resources the process is currently holding are preempted.

→ Alternately ⇒ If a process requests some processes and if they cannot be allocated and is allocated to some other process i.e waiting for additional resources then we can preempt the resources from the waiting process and allocate it the requesting process.

⇒ In order to remove the circular wait condition we can impose total ordering on all the resources and to require that each process request resources in an increasing order of enumeration.

→ Let us consider a process initially request for resource R_i after that process requests other resource R_j iff $F(R_j) > F(R_i)$

→ Alternatively we can require that a process requesting an instance of process R_j must

have release any process R_i iff $f(R_i) \geq f(R_j)$